# Alleviating Scalability Issues of Checkpointing Protocols

Rolf Riesen*, Kurt Ferreira†, Dilma Da Silva‡, Pierre Lemarinier*, Dorian Arnold§ and Patrick G. Bridges§

*IBM Research - Ireland
rolf.riesen@ie.ibm.com
pierrele@ie.ibm.com

†Sandia National Laboratories
Albuquerque, USA
kbferre@sandia.gov

‡IBM Research - Watson, USA
dilma@acm.org

§University of New Mexico
Albuquerque, USA
darnold@cs.unm.edu
bridges@cs.unm.edu

***Abstract*—Current fault tolerance protocols are not sufficiently scalable for the exascale era. The most-widely used method, coordinated checkpointing, places enormous demands on the I/O subsystem and imposes frequent synchronizations. Uncoordinated protocols use message logging which introduces message rate limitations or undesired memory and storage requirements to hold payload and event logs. In this paper we propose a combination of several techniques, namely coordinated checkpointing, optimistic message logging, and a protocol that glues them together. This combination eliminates some of the drawbacks of each individual approach and proves to be an alternative for many types of exascale applications. We evaluate performance and scaling characteristics of this combination using simulation and a partial implementation. While not a universal solution, the combined protocol is suitable for a large range of existing and future applications that use coordinated checkpointing and enhances their scalability.**

## I. Introduction

Fault tolerance for parallel computation has been explored for several decades and many algorithms and protocols have been proposed and evaluated since then. Among them, coordinated checkpointing has been the most commonly used because of its simplicity, while uncoordinated checkpointing with sender-based message logging has been considered the most efficient and scalable.

Entering the exascale era, some of the fundamental assumptions underlying the research of the last thirty years are changing. Therefore it is important to reevaluate fault tolerance techniques in light of these new conditions. Faults are expected to be frequent, and multiple simultaneous failing components common. On a per core basis, access to distributed, stable storage will become more expensive in terms of latency and bandwidth. That means that protocols relying heavily on stable storage may not scale. The per-core amount of memory in an exascale system will be small compared to today's memory sizes per CPU and that memory has to be shared between the application and the message logs. Finally, some applications require high message rates and will be severely hampered by protocols that limit send rate.

Past approaches have often been evaluated on very small systems and small problem sizes. Certain exascale effects, such as a higher system-wide failure rate or higher per-core cost of access to stable storage, have not been adequately taken into consideration. It is necessary to do that now, review existing approaches, and enhance them to become suitable for exascale.

Coordinated checkpointing is relatively easy to implement, especially for self-synchronizing applications. However, increasing machine size and the higher number of expected faults in an exascale system, make coordinated checkpointing less and less efficient.

Sender-based message logging in conjunction with uncoordinated checkpointing addresses some of the drawbacks of coordinated checkpointing. However, because of the scale of future systems and application characteristics, message logging alone is not a complete answer either.

A fault tolerance protocol for exascale based on checkpointing techniques must address these challenges:

1) Message payload logs can grow large and take valuable memory away from computation.
2) Because recoveries will become more frequent, recovery times have to be quick and efficient.
3) Multiple simultaneous faults must be tolerated.
4) Message send rate is important. Waiting for an acknowledgment for events written to stable storage is too costly.

In this paper we investigate a combination of message logging with coordinated checkpointing. The two methods support each other in creating a system that has fewer whole application restarts by using message logs to restart individual processes in most cases. This avoids "getting stuck" in application restart because one or another process keeps failing. Coordinated checkpointing in turn assists message logging by (somewhat) limiting the log sizes and providing a fall-back in cases where the system has gotten into an inconsistent state.

This combination of existing technologies and tailoring the resulting algorithm to exascale addresses the challenges listed above. We address 1) above with coordinated checkpointing, at that time the payload and event logs can be cleared. Furthermore, the combined algorithm does not assume that the logs reside in stable storage.

This research was partially supported by an Enterprise Partnership Scheme grant co-funded by IBM, the Irish Research Council for Science, Engineering & Technology (IRCSET), and the Industrial Development Agency (IDA) Ireland.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

For 2) above we reduce the recovery time because most failures cause a single process restart rather than a costly coordinated, full-application restart. Addressing 3), the enhanced protocol tolerates any number of simultaneous faults and degenerates to coordinated checkpointing if necessary. Finally, for 4), we can use an optimistic logging protocol which does not impose delays between consecutive sends.

This paper makes the following contributions:

- Describes a combined fault tolerance technique that is enhanced and adapted for exascale class systems.
- Evaluates, using simulation and an initial implementation of this algorithm, the scalability of the technique under various fault conditions and assumptions.
- Provides estimates on how many spare nodes are required to limit the number of full application restarts.
- Discusses the relative merits of various approaches to fault tolerance at exascale.

## II. Design

Our proposed algorithm combines coordinated checkpointing with optimistic message logging. It uses additional nodes as loggers to store event logs and as spares to be swapped in when nodes fail. None of the components – senders holding payload logs, loggers keeping events, nor spare nodes – are assumed to remain fault free during the execution of a parallel job. Only the storage system is assumed to be stable. Although deceptively simple at a high-level, combining these algorithms in this manner requires a protocol described in the following sections.

### A. Definitions and assumptions

There are three classes of nodes: compute nodes ($Sc$) send and receive messages, additional nodes set aside as spares ($Ss$) to replace failed nodes, and a few more nodes, called loggers ($Sl$), which are used to store message event logs. Therefore, our algorithm requires $N = |Sc| + |Ss| + |Sl|$ number of nodes to execute an application that runs on $n = |Sc|$ compute nodes.

Before we describe the algorithm in detail, we list the underlying assumptions:

- Communication channels observe FIFO order.
- Fail-stop model, which implies for instance that messages are delivered correctly, or not at all.
- Any number of compute nodes, loggers, and spare nodes can fail at any time during a job's execution.
- Neither the message payload logs nor the event logs are assumed to be stored in stable storage.
- The system can be modeled as an asynchronous system with a perfect failure detector. Moreover, we assume the presence of a Reliability, Availability, and Serviceability (RAS) system for detecting failures and restarting processes on spare nodes we have set aside.
- Process checkpoints are stored in stable storage.
- One MPI process per node and only MPI message reception can cause a node to behave non-deterministically.

This last assumption does not mean an application can use only one process or thread per node, nor that scheduling needs to be deterministic. What is required is that all externally visible events are deterministic and solely determined by the order and content of incoming MPI messages. Local non-determinism is fine as long as it does not cause externally visible non-deterministic events.

### B. Protocol description

The protocol makes use of blocking coordinated checkpoints as described in [1] and [2]. A coordinated checkpoint can be triggered at the time applications enter a synchronization point or upon request by the RAS system. We call the time between two consecutive coordinated checkpoints a phase. The protocol also relies on optimistic message logging similar to [3]. A small, constant size information field is piggybacked onto each message. That information is used to uniquely identify a message. It is also used to determine, at the time a process restarts after a failure, if the optimistic protocol "failed" and a full application restart from the last global checkpoint must be triggered due to possible inconsistency.

*1) Variables:* Each process $p$ in the system maintains three counters or local logical clocks. The first counter, $S_p$, increases for each message sent by $p$. The second counter, $R_p$, increases for each message $p$ receives. The last counter, $rest_p$, is incremented each time $p$ restarts.

Each process $p$ also keeps four sparse arrays: $AS_p[]$, $AR_p[]$, $ARestart_p[]$ and $ASent_p[]$, all indexed by MPI rank. $AS_p[]$ and $AR_p[]$ store the last $S_{p_{id}}$ clock and $R_{p_{id}}$ clock $p$ has seen from process $p_{id}$. $ARestart_p[]$ stores the current restart counter of every process. Finally, $ASent_p[j]$ stores the ID of the last message sent to process $P_j$.

*2) Normal execution:* Every process is either in its normal execution state or in restart mode. During normal execution, when process $p_i$ sends a message to $p_j$, it increases its clock $S_{p_i}$ and piggybacks three counter values onto the message: $rest_{p_i}$, $S_{p_i}$ and $R_{p_i}$. Note that, contrary to causal protocols, the amount of information piggybacked is small and of constant size. The payload of a message and this piggybacked information is stored in a log in the sender's memory and is indexed by $p_j$, $S_{p_i}$. Finally, the process checks whether $S_{p_i} > ASent_{p_i}[j]$. If true, it sends the message, else it does not and continues its execution.

When process $p_j$ receives a message from $p_i$, containing $rest_{p_i}$, $S_{p_i}$ and $R_{p_i}$, it first checks whether $rest_{p_i} = ARestart_p[i]$. If not it drops the message, waiting for a new message to be sent after the sender has restarted. Else it processes the message. It increases its own clock $R_{p_j}$, and it updates its arrays: $AR_{p_j}[i] = max(AR_{p_j}[i], R_{p_i})$, $AS_{p_j}[i] = max(AS_{p_j}[i], S_{p_i})$. It then sends an event containing information about the sender and receiver of this message to its logger. That event is small and of constant size as it contains only ($p_i$, $S_{p_i}$, $R_{p_i}$, $p_j$, $R_{p_j}$). $p_j$ finally delivers the message and optimistically continues its execution without waiting for an acknowledgment from the logger. Each event logger serves many compute nodes and erases its log at the beginning of each phase. Message payloads are not stored in event loggers.

When process $p_i$ receives $rest_{p_k}$ from the RAS system to inform it of a process $p_k$ restarting, it updates $ARestart_{p_i}[k]$ = $rest_{p_k}$ and $ASent_{p_k}[i]$ = 0. It sends $m_rest_{p_i}$ ($AS_{p_i}[k]$, $AR_{p_i}[k]$) to $p_k$ to let the restarting process know which message between $p_i$ and $p_k$ was the last one successfully delivered. Finally $p_i$ resends every message kept in its logs that was sent to $p_k$ before it failed during this phase. These messages are resent using the payload and piggybacked information contained in the sender's log, with only the value $rest_{p_i}$ updated to the current value of the sender.

*3) Failure handling:* If a logger fails, the application continues to execute. If it reaches a global checkpoint, the RAS system uses a spare node to assign a new logger and informs every process about it. When the RAS system detects process $p_k$ has failed, it checks if a spare node is available. If no spare node is available, the RAS system forces an application restart. In that case the job scheduler can be involved to reschedule the application at a later time when more nodes are available. If a spare node is available, the RAS system restarts $p_k$ on that spare node, sends to every process the incremented $rest_{p_k}$ and sends the array $ARestart[]$, listing the current restart counter of every process, to $p_k$.

*4) Process restart:* When process $p_k$ restarts, all its arrays and counter are zeroed. It confirms with the RAS system that its logger is alive and asks for the list $\mathcal{L}$ of all events logged for $p_k$. In case its logger has failed, the whole application has to be restarted by the RAS system.

For each event ($p_i$, $S_{p_i}$, $R_{p_i}$, $p_k$, $R_{p_k}$) $\in \mathcal{L}$, it updates $AR_{p_k}[i]$ = $max(AR_{p_k}[i]$, $R_{p_i})$. It then receives and updates $rest_{p_k}$ and $ARestart_{p_k}[]$ from the RAS system. Finally $p_k$ waits for $m_rest_{p_i}$ = ($AS_{p_i}[k]$, $AR_{p_i}[k]$) from every process $p_i \neq p_k$. When it receives one, $p_k$ updates $ASent_{p_k}[i]$ = $AS_{p_i}[k]$ and checks that $AR_{p_i}[k] \leq |\mathcal{L}|$. If this is not the case, it asks the RAS system to trigger a full application restart, since this indicates that one or more events did not get logged.

When $p_k$ has all $m_rest$ messages, it starts execution in restart mode. Restart mode is similar to normal mode with one difference: when a message has to be received, the first element ($p_i$, $S_{p_i}$, $R_{p_i}$, $p_k$, $R_{p_k}$) is popped from $\mathcal{L}$ and $p_k$ selects or waits for the message sent by $p_i$ containing ($S_{p_i}$, $R_{p_i}$). Normal execution continues when $\mathcal{L}$ becomes empty.

### C. Case studies

We present a few scenarios of execution to highlight features of the protocol presented in the previous section. Fig. 1 shows the basic operations of the algorithm. In this example, four processes, one logger, and a spare node progress through two phases without incident. After the second barrier, $b_2$, fault $f_1$ occurs and process $P_3$ has to restart.

The process restart is done on the spare node. The last checkpoint has to be read, the work of the entire phase has to be redone, and the next checkpoint has to be written before process $P_3$ can join the other processes in barrier $b_3$. Once a process has written its own checkpoint and is waiting in the barrier $b_3$, it will be restarted into this barrier should that process fail at the end of the phase, using the checkpoint it just did. If a global application restart is triggered, then all processes will restart from the checkpoint of barrier $b_2$.

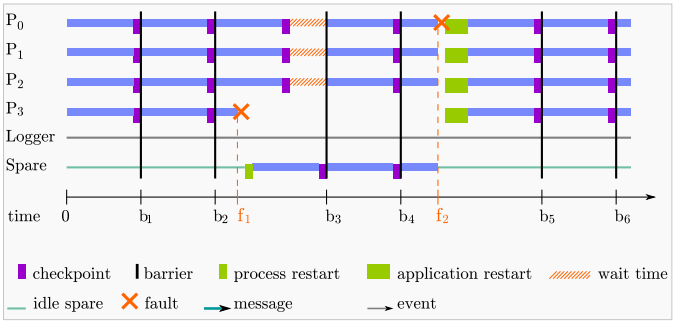


Fig. 1. Progress of a parallel application.

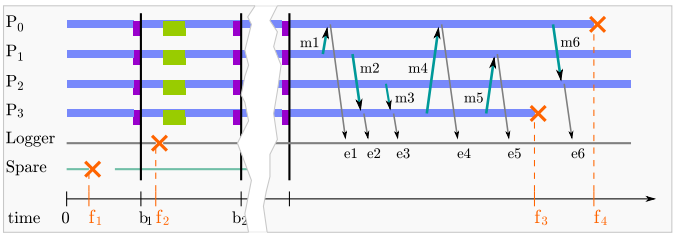


Fig. 2. Several examples discussed below.

In our example, processing continues until the second fault occurs at $f_2$. At that time, no more spare nodes are available and the entire application is forced to restart. Because a full application restart places more demand on the I/O system, it will take longer than the restart of an individual process. We assume that after an application restarts, enough nodes are available again to host the compute processes, loggers, and spare nodes for the next phase.

More examples are in Fig. 2. If a spare node fails, at $f_1$ in the graphic, an attempt is made to reboot it. In our simulations we assume that it will succeed 50% of the time and "fix" the problem that caused the fault. In that case the spare node becomes available again. If a compute node or a logger fail, we attempt the same trick and, if successful, place the freshly booted node into the pool of spares.

The second example in Fig. 2 is at $f_2$, when the logger fails. There is no need to restart the whole application at this stage. Only if a node fails which needs event information from that logger, would an application restart become necessary. In our simulation we decided to ignore this optimization though, and do a full restart a soon as a logger fails.

Three more examples using Fig. 2: 1) Assume that messages $m_1, \ldots, m_6$ on the right hand side of the figure have been received, and that the corresponding events, $e_1, \ldots, e_6$, have been recorded by the logger, then the complete event log would look like Table I. This log could be split among different loggers, but every event sent by a particular process is stored on its assigned logger.

When fault $f_3$ occurs in Fig. 2, process $P_3$ needs to restart. At the beginning of its restart phase, it will ask its logger for all event entries that carry information about its previous receives.

TABLE I
AN EVENT LOG EXAMPLE

| Event | Source | | | Destination | |
|---|---|---|---|---|---|
| | rank | snd | rcv | rank | rcv |
| $e_1$ | $P_1$ | 1 | 0 | $P_0$ | 1 |
| $e_2$ | $P_1$ | 2 | 0 | $P_3$ | 1 |
| $e_3$ | $P_2$ | 1 | 0 | $P_3$ | 2 |
| $e_4$ | $P_3$ | 1 | 2 | $P_0$ | 2 |
| $e_5$ | $P_3$ | 2 | 2 | $P_1$ | 1 |
| $e_6$ | $P_0$ | 1 | 2 | $P_2$ | 1 |

In our example, those are events $e_2$ and $e_3$, thus the logger will build the list $\mathcal{L} = (e_2, e_3)$. Armed with this knowledge, $P_3$ determines that it received messages $m_2$ and $m_3$ in that order from $P_1$ and $P_2$. $P_3$ will then wait for information from the other processes and will discover that $P_0$ and $P_1$ have received messages from $P_3$ that do not need to be sent again. $P_3$ will receive $m_2$ and $m_3$ in some order. It will re-execute and when receiving the first message, then $m_2$ will be delivered first, followed by $m_3$, according to $\mathcal{L}$. $P_3$ will fake emission of $m_4$ and $m_5$ and store their contents in its message log.

2) Assume that event $e_3$ got lost or delayed and is not available in the log when $P_3$ restarts. This could happen because $P_3$ sends event $e_3$ followed immediately by $m_4$ to its logger and $P_0$. For one reason or another, $e_3$ is delayed and held in $P_3$'s resend buffer, while $m_4$ is successfully received by $P_0$. Since communication channels are FIFO, only a suffix of the sequence of events logged regarding a process can be lost. For instance, if $e_4$ is received by the logger of $P_0$, then $e_1$ has been received and logged.

After $P_3$ restarts, it will learn about event $e_2$, but not $e_3$ and will thus build $\mathcal{L} = (e_2)$. From $\mathcal{L}$, $P_3$ deduces it has previously received a single message and could decide to receive a different message than $m_3$ following $m_2$. This would lead to an inconsistent execution regarding $P_0$. There are two possibilities at this stage: $P_0$ has effectively received message $m_4$ when getting informed of $P_3$ restarting. In that case it will send the corresponding information; i.e., $m_rest_{P_0}(1, 2)$. $P_3$ deduces from this message that it should have gotten at least two events from its logger and requests a full application restart. In the case of $P_0$ not having received $m_4$, it will send $m_rest_{P_0}(0, 0)$ when $P_3$ restarts. $P_0$ then deduces its restart is legitimate and continues its execution. In that case, consistency is enforced within $P_0$ by detecting when receiving $m_4$ that the message was sent before $P_3$ restarted, using $rest_{p_3}$, and by dropping this message.

It is important that a sender replays all of its messages in the same order as before the restart. In our protocol, either all non-deterministic events were logged to the sender's logger or the receiver will be able to tell the sender the number of events it handled prior to this communication. This can be used as a hint of the necessity to do a full application restart.

3) For the case of multiple failures, assume $P_3$ and $P_0$ fail as in Fig. 2. The case of $e_4$ not being logged, potentially creating an inconsistency for $m_6$, has already been described in 2) above. It will either trigger a full restart, or any message can be received instead of $m_4$ without creating inconsistency. Thus, let us assume $e_4$ is logged. In that case it may be that $P_3$ is also restarting and is not yet able to reproduce $m_4$ in exactly the way it did in the previous phase. This can happen if $e_3$ has been lost. When $P_3$ restarts, either $P_0$ has not failed yet in which case we are in a similar situation as described in 2) and the same logic applies that triggers a full restart. Or, $P_0$ already restarted and it has received the list of events from its logger, holding $e_4$ from which it recovered the knowledge that $P_3$ performed two receives before sending $m_4$. It will deliver this information to $P_3$ when it restarts, making $P_3$ detect the missing event $e_3$ and triggering a full application restart.

We expect these scenarios to be very rare occurrences because supercomputer networks seldom drop small events but succeed in delivering subsequent application messages from the same node.

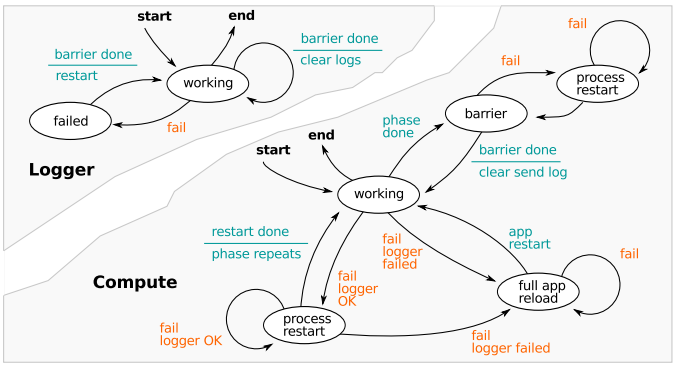


Fig. 3. Logger and compute node state diagrams.

## III. SIMULATION

We wrote a discrete event simulator to evaluate the scalability of the combined protocol. We briefly discuss the implementation and then present the results we have obtained.

### A. Implementation

Fig. 3 shows the possible states of a message event logger and a compute node. A logger is either working and logging messages, or it has failed and been swapped out. If so, it can no longer assist failed compute nodes, because it has lost the message events since the current phase began. It will resume a working state after the next barrier, or cause an application restart, if a compute node needs log information.

The basic mode of a compute node is working until the checkpoint interval time has elapsed, then write a checkpoint, and enter a barrier, where it waits for the other nodes. If a compute node fails, it performs a single-node restart at the beginning of the current phase. If a compute node's logger has also failed, a single-node restart is no longer possible and the whole application must restart. Once a node reaches its barrier, future faults will restart it into that barrier, restarting from the most recently written checkpoint.

The simulator is configured with $N = |Sc| + |Ss| + |Sl|$ nodes, each executing the state diagram in Fig. 3 according to

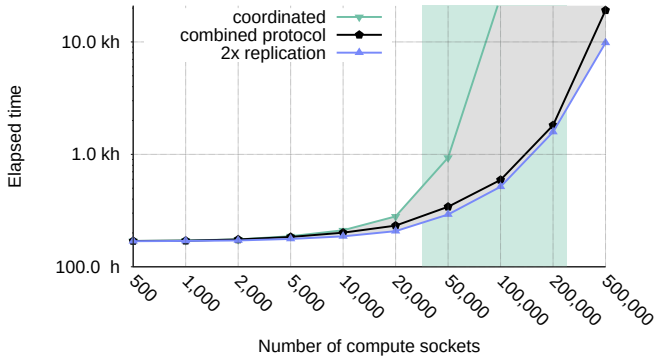


Fig. 4. Elapsed time for a 168-hour job, 5-year socket MTBF.

its role. Spare nodes can be used to replace failed loggers or compute nodes. When the pool of spare nodes is exhausted, an application restart becomes necessary. Spare nodes, while waiting for assignment, can also fail. When a node fails, an attempt is made to reboot it. With a probability of 50% this succeeds in "fixing" whatever caused the node to fail, and the node is placed into the spare pool five minutes later.

For all the simulations in this paper we ran a 168-hour, weak scaling job. Each of the $|Sc|$ compute nodes needs to get 168 hours (1 week) of work done. Because of checkpoint overheads and restarts due to faults, the elapsed time for each node to complete 168 hours of work is larger. We ran each experiment until the elapsed time was within $\pm 5\%$ of the 95% confidence interval. For most experiments that was five trials; our minimum number for any experiment.

### B. Results

Fig. 4 shows how much time is needed to complete a 168-hour job. The top curve in the plot, labeled "coordinated", is a calculation of how much time a job requires when it uses only coordinated checkpoint/restart for fault tolerance. We use Daly's Equation 20 from [4] to calculate the elapsed time. We use Equation 37 from the same paper to calculate the optimal checkpoint interval $\tau_{opt}$.

The bottom curve in Fig. 4, labeled "2x replication" is a model of running an application on a system with dual redundancy. We use Equation 1 from [5] for that. The gray band between these two curves represents our area of interest. We know coordinated checkpointing does not scale well. That curve, therefore, represents an upper bound. In [5] the authors found that dual redundancy greatly reduces the number of faults an application experiences, albeit at twice the hardware cost. We use it as a lower bound in our comparisons and a goal to strive for. The black middle curve in Fig. 4, is the simulated performance of the protocol described in this paper. With the parameters chosen for the experiment in Fig. 4, it performs very well compared to coordinated checkpointing.

The two models and our simulator require input parameters. One is the number of compute nodes $|Sc|$ available to the application. Redundant computing requires $N = 2 \cdot |Sc|$ nodes and the combined protocol requires logger and spare nodes. Our plots always show $|Sc|$, not $N$. For an exascale system to be built near the end of this decade, it is usually assumed that it will have between 30,000 and about 300,000 nodes, depending on configuration [6]. We assume each node hosts a single MPI process, possibly with multiple threads. The shaded area in Fig. 4, and in other plots in this paper, indicates the expected design space for future exascale systems. The socket MTBF for this example is 43,800 hours (5 years). That number is based on [7] and [8] and includes everything – not just hardware – that could cause a process on a socket to fail; e.g., software, external power and cooling, human error, etc.

The time required to read and write a checkpoint depends on many factors, including its size, the I/O bandwidth, and contention for network paths and storage units. We arbitrarily set the amount of data each node saves and restores to 16 GB. We further assume that a single node can read or write that amount of data in 3.2 seconds; i.e., it can transfer at 5 GB/s.

For a full application checkpoint or restart, we assume that the parallel file system has an aggregate bandwidth of 0.5 TB/s, 1.0 TB/s, 5.0 TB/s, 10 TB/s, or 30 TB/s. These values cover the range expected by [6]. The available bandwidth has to be shared by the number of nodes in the system. For example, in a 100,000-node system capable of an aggregate I/O bandwidth of 5.0 TB/s, each node writing its 16 GB of data would take 320 seconds; i.e. I/O may be limited by node or aggregate bandwidth.

We use Daly's $\tau_{opt}$ [4] to set the checkpoint interval. For the redundant computing model and the simulation of the combined protocol, that is not entirely correct, since $\tau_{opt}$ is meant for coordinated checkpointing. However, it gives us reasonable starting points for our experiments.

For the simulations in this paper, we used exponentially distributed faults assuming nodes fail independently. Not enough is known yet about failure distribution and their interdependence in future systems.

Fig. 5 analyzes the impact of MTBF on elapsed time. We perform the same experiment as in Fig. 4 and steadily increase the socket MTBF to 100 years. The three planes in this plot represent, from the top, coordinated checkpointing (green), the combined protocol of this paper (gray), and redundant computing (blue). The elapsed time for coordinated checkpointing at large socket counts and small MTBF, exceeds the bounding box of Fig. 5.

At small node counts and large MTBF (left corner of Fig. 5) the elapsed time for all three methods is about the same. As we increase the number of nodes, execution times using redundant nodes or the combined protocol, stay close together, with redundant computing needing always less time to complete than the other approaches. Coordinated checkpoint/restart is several times slower than the other two approaches at high node counts (front center corner of Fig. 5). When we decrease the node MTBF, the situation for all approaches gets worse, but much more so for coordinated checkpointing (right side corner of Fig. 5).

Redundant computing uses additional hardware to reduce the number of faults visible to an application: $N = 2 \cdot |Sc|$.

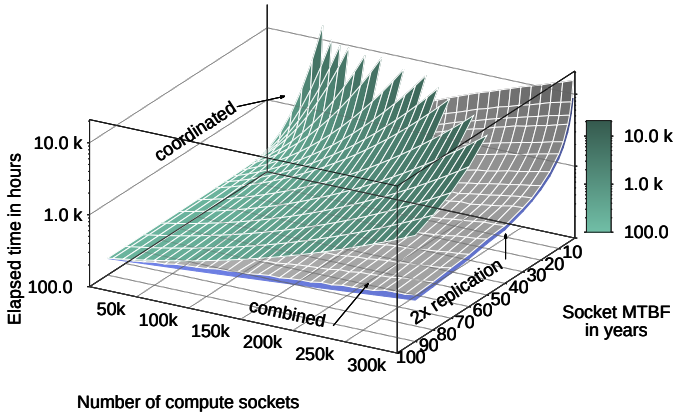


Fig. 5. The elapsed time of a process varies with the number of nodes and the socket MTBF of the system as well as the algorithm used.

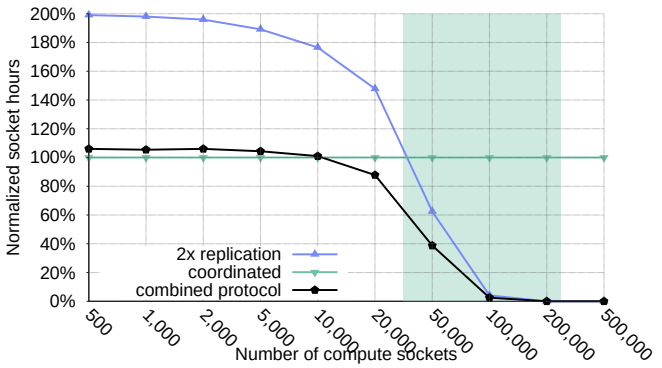


Fig. 6. Socket hours for a 168-hour job (5-year socket MTBF).

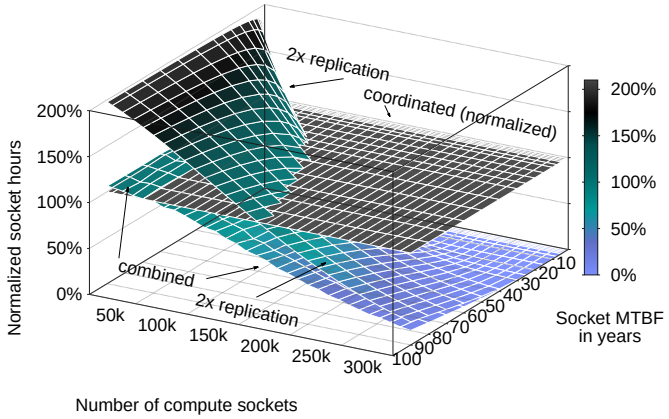


Fig. 7. Socket hours needed to complete a 168-hour job for various MTBF.

Our combined method uses additional nodes for loggers and spares. For our experiments we set the number of loggers to $|Sl| = 0.01 \cdot |Sc|$ and provided $|Ss| = 0.05 \cdot |Sc|$ spares, for a total of $N = |Sc| + 0.05 \cdot |Sc| + 0.01 \cdot |Sc|$ nodes. We will see later that 5% spare nodes is plenty. The 1% logger nodes is a compromise between fault tolerance and performance. Fewer logger nodes in a system is better for fault tolerance, since a failure of any of them usually mandates an expensive application restart. However, too few logger nodes can create a bottleneck in writing and retrieving message events.

In Fig. 6 we use socket hours as a metric to measure hardware costs of a fault tolerance method. It is the number of sockets required to carry out a computation multiplied by the total time to completion. We normalize the results against coordinated checkpointing since that method uses the fewest number of nodes. At low node counts, redundant computing uses 200% the number of nodes. With higher node counts, redundant computing uses less time to finish than the same size job using coordinated checkpointing. Therefore, the ratio decreases and, beyond 20,000 nodes, redundant computing needs fewer socket hours than coordinated checkpointing to complete a job of the same size. Our protocol also starts out using more resources, but much fewer than redundant computing. Beyond 10,000 nodes, its socket hour consumption is becoming much less than either redundant computing or coordinated checkpointing. At very high node counts, redundant computing and the combined protocol are much better than coordinated checkpointing.

The shaded area, marking the expected design space of future exascale systems, helps us see that redundant computing may not be cost effective below the exascale range, while checkpointing methods like the one presented in this paper, can play a role in large scale systems, even if they are smaller than the top-ranked systems.

Again, the node MTBF plays an important role in this evaluation. We repeat the experiment for Fig. 6 for a range of MTBFs in Fig. 7. The middle (gray) plane represents coordinated checkpointing, which we normalize at 100%. The lower plane is the combined protocol. It starts out, at low node counts and high MTBF (left corner of the plot), above the gray plane, indicating that it requires more socket hours than coordinated checkpointing. As the number of nodes increases and the MTBF drops, the combined protocol results fall below the gray plane, indicating better resource usage.

Redundant computing starts out high in the left side corner of Fig. 7. It, too, drops below the gray plane representing coordinated checkpointing, but at a later point than the combined protocol. That dip towards better resource utilization occurs sooner if the MTBF is low (far back corner of Fig. 7.)

In a system with a higher aggregate I/O bandwidth, checkpoints can be written more quickly. This allows coordinated checkpointing to scale better. For Fig. 4 we used an aggregate I/O bandwidth of 0.5 TB/s. We repeated the experiment with an I/O bandwidth of 30 TB/s. The result is shown in Fig. 8. With 0.5 TB/s aggregate bandwidth, there are clear advantages for redundant computing and the protocol presented in this paper, starting at about 20,000 nodes. If the bandwidth is as high as 30 TB/s, more than 100,000 nodes are needed before the overhead of coordinated checkpointing causes much longer execution times than the two other fault tolerance methods.

I/O bandwidth to storage is improving as technology marches forward but is not growing at the same rate as processor performance. However, enormous aggregate bandwidths, such as 30 TB/s, will not be cheap and the cost of that will have to be taken into consideration when deciding on a fault tolerance solution. A compromise between an affordable I/O

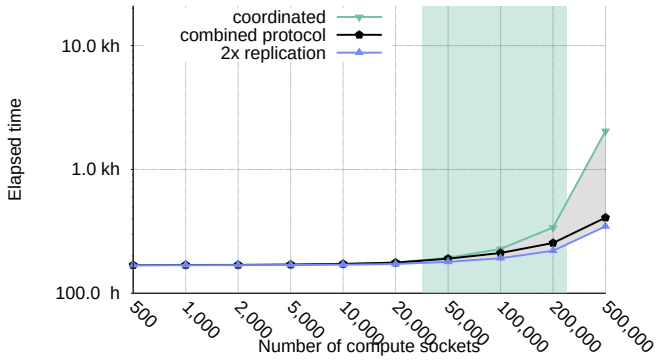


Fig. 8. Same as Fig. 4 but with aggregate I/O of 30 TB/s instead of 0.5 TB/s.

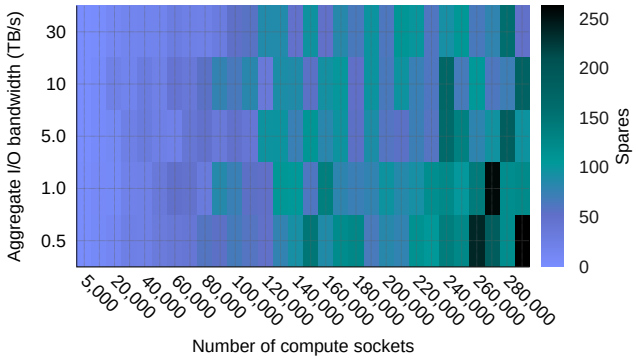


Fig. 10. The number of spares needed to avoid an application restart.

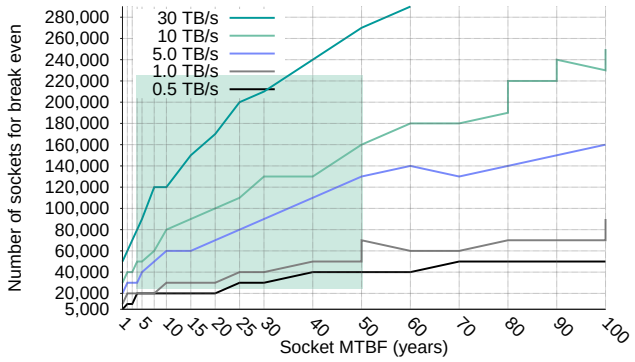


Fig. 9. The number of sockets at which point coordinated checkpointing alone uses fewer socket hours than the combined protocol. Above a given line, the protocol described in this paper may be preferable.

system and an efficient fault tolerance method has to be found.

Given these trade-offs, it is interesting to see where in the parameter space of projected exascale systems, the break-even point lies. I.e., how many compute nodes are required before the combined fault tolerance protocol of this paper completes a job in fewer socket hours than a system that uses coordinated checkpointing alone.

Fig. 9 shows the break-even lines for five different aggregate I/O bandwidths. Given a node MTBF, a node count above the break-even line indicates that the protocol of this paper may be preferable over coordinated checkpointing. Like in previous graphs, the shaded area indicates parameter ranges in which an exascale system in the next few year may be built.

Although we used 5% spare nodes for all of our simulation runs, that many spares are actually not needed. Fig. 10 shows a typical example. For various aggregate I/O bandwidths ($y$-axis) and increasing number of nodes ($x$-axis), we use color shading to indicate the largest number of failed or in-use spares at any time during the simulation. A darker shade means more spares were needed. All our results, including the 70-year node MTBF example of Fig. 10, show that more spares are needed with higher node counts (right side of graph), and lower aggregate I/O bandwidth (bottom part of graph). Not surprisingly, the lower the MTBF, the more spares are needed. However, in almost 3,000 simulations over a wide range of aggregate I/O bandwidths, MTBFs, and node counts, even with a 1-year node MTBF, the largest number of spares needed was 1309 in a 270,000-node experiment.

## IV. Library Implementation and Testing

To evaluate the runtime performance and scalability of this design, we implemented the combined protocol described here as a library, called *ml*MPI. It features:

- A configurable number of ranks set aside as logger nodes.
- Send payload data tracked and saved on the local node in host memory.
- ID information in all point-to-point messages.
- An event sent to a logger node for each receive.
- Emulated local checkpoints, failures, and restarts.

Our implementation is at the MPI profiling layer. We chose that for quick prototyping and portability to the majority of modern MPI libraries. Implementing at the PMPI layer represents a worst case overhead for this design. A full implementation needs to be at a lower layer in order to optimize the handling of some protocol messages more efficiently. This initial implementation has a number of limitations. Most importantly, collective operations are not fully implemented and can not fully recover from an actual failure. Due to this, restarts do not currently pull messages from corresponding nodes and instead busy-wait locally for a time equal to the interval since the last local checkpoint, simulating the rework time. *ml*MPI is in the approval stage for open source release.

### A. Results

Due to the limited availability of dedicated time on large-scale systems, only limited runtime results using *ml*MPI are available currently. We use simulation in Section III for large scale results, and the library to evaluate runtime overheads for a number of micro-benchmarks: latency, bandwidth, and message-rate. We measure the expected log growth rates, a key scalability issue for logging based protocols, for four key HPC workloads: CTH [9], SAGE [10], LAMMPS [11], [12], and HPCCG [13]. We do not show the runtime results for these workloads because at the node counts tested, up to 128 nodes, we saw no difference in run times when compared to the native MPI stack.

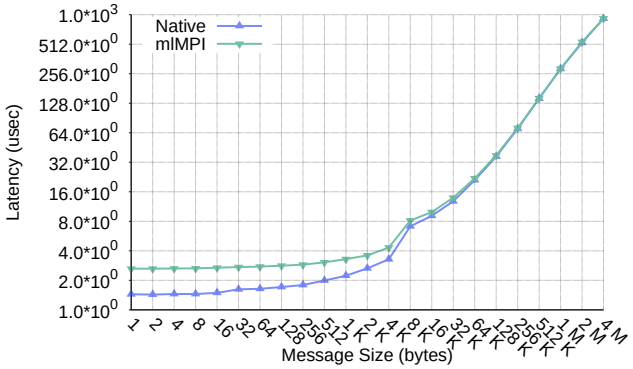


Fig. 11. MPI latency performance comparison of *ml*MPI library to native.

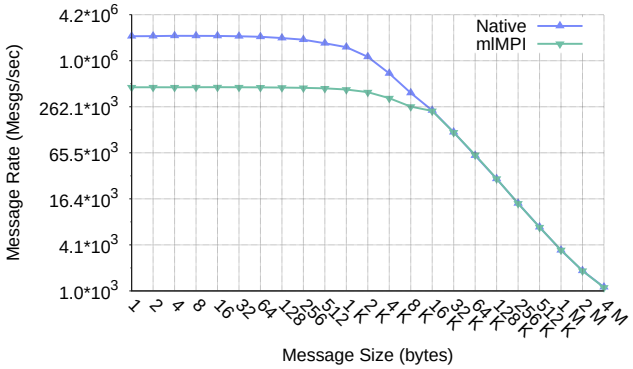


Fig. 12. MPI message rate performance comparison.

These applications represent a range of computational techniques, are frequently run at very large scales, and are key simulation workloads for both the US DOD and DOE. These four applications represent different communication characteristics and compute to communication ratios. Therefore, the overhead of *ml*MPI affects them in different ways.

All tests were run on a small Cray XE6 system, the same architecture as the ACES Cielo platform. Each node consists of a Dual socket AMD Opteron 6136 eight-core Magny-Cours processor @ 2.4 GHz with 32 GB of main memory and Cray's Gemini proprietary network.

*1) Micro-benchmark performance:* As micro-benchmarks do nothing but constantly transmit and receive messages within the system, we expect them to exhibit higher overheads than a full HPC application. The latency measurements in Fig. 11 show the impact of the message logging protocol. Small-message bandwidth also decreases a little bit due to the increased latency. We see that for smaller message sizes, as expected, performance is impacted. The majority of this performance loss is due to the additional message ID information that must be included in every message. At the cost of portability, a message logging library built at a lower level within the network stack could greatly increase performance. This is cost we are currently willing to pay in order to maintain portability. A small amount of performance is lost in the library due to the pessimistic send log commit protocol that is used locally on a node. As the logs are not pushed to stable storage, an optimistic protocol could easily be used in its place at the cost of additional bookkeeping within the library.

Fig. 12 illustrates the MPI message rate performance for our library in comparison to native. Message rate is the most severely impacted of the three micro-benchmarks. Again, the performance loss is attributed to the transmission of message IDs. We are working on methods to reduce this message rate slowdown without sacrificing portability.

*2) Message log growth rates:* In the previous section we showed that the overhead of *ml*MPI can be quite large for micro-benchmarks in comparison to native. In contrast, in our limited scale testing, our applications have shown no runtime performance impact. We expect these overheads can be kept low as long as we scale the number of logging nodes along with application nodes.

TABLE II
PER MPI PROCESS LOG GROWTH RATES.

| Application | Log growth rate per MPI process |
|---|---|
| HPCCG | 0.6 MB/s |
| LAMMPS | 1.5 MB/s |
| SAGE | 13 MB/s |
| CTH | 40 MB/s |

Along with runtime performance, we also measured the log growth rate for our example workloads. This growth rate is important as the logs in our library share memory with the application. The log growth rate is tracked with the library and the initial problem setup and tear-down and is not included as these messages are typically much larger than messages sent during the main compute portion of the application. Table II shows the per-process log growth rate per second. These numbers represent the averages for each application between 2 and 1024 MPI processes. After 128 ranks the per-process log size remained relatively constant for all applications. These log growth rates can vary dramatically; from 40 MB/second for the bandwidth sensitive application CTH, to 0.6 MB/second for the communication avoiding HPCCG.

## V. DISCUSSION

Our simulation studies indicate that a combined approach, like the one we present in Section II, can be used to take proven fault tolerance methods and make them more suitable for exascale. We compare our approach to coordinated checkpointing alone, because it is a crucial part of our combined protocol and is a widely used method. At the other end of the extreme, we look at redundant computing which greatly reduces the number of faults visible to an application.

There are several reasons why our protocol works well at large scale. The most vulnerable parts of our system are the logger nodes. Most of the time when one of them fails, an application has to perform a full restart. However, since there are relatively few loggers required, the probability of one of them failing is small. Another reason we scale well is our optimistic approach to message logging. Nodes neither

get nor wait for an acknowledgment when writing logs. Therefore, these operations can be done asynchronously in the background without pausing an application. Furthermore, when the optimistic approach creates an inconsistency, our protocol can detect that as soon as a node restarts and then request a full application restart. It never needs to roll back further than the last complete full application checkpoint.

Our experiments show a measurable impact on micro-benchmark performance. This is due in part to our non-optimized initial implementation, but has negligible impact on real application performance. In addition, the overhead our protocol introduces, for example the three counters we send in each message, is of constant size and has no scaling limitations. Our approach is most suitable for extreme-scale application which already use coordinated checkpointing, or self synchronize on a regular basis. Like all message logging based protocols, there is an issue with storing the message payload logs. Our protocol does not require them to be written to stable storage, which lessens the demand on the I/O subsystem. Nevertheless, the size of these logs stored in memory that could be used by applications, is of concern.

Due to our regular checkpoints, the size of these logs can be limited. This is a trade-off controlled by number and size of messages sent, and the checkpoint interval. A large checkpoint interval reduces demand on the I/O subsystem and lowers checkpoint overhead. A small checkpoint interval increases overhead, but reduces the amount of memory needed to store message payloads. We also looked at the impact of aggregate I/O bandwidth. A high aggregate bandwidth is beneficial and extends the scalability of all three approaches, but coordinated checkpointing in particular. The trade-off is the high cost of such a storage system. It requires a massively parallel I/O subsystem coupled with a very fast network capable of carrying a huge volume of data.

Another approach is to make the system more reliable. For several reasons, this is unlikely to happen. Near-future exascale system will be made of components that are the same or similar to the ones used in standard compute technology. Manufacturers are struggling to keep reliability of these components at today's levels because feature sizes and supply voltages are shrinking, making these devices more vulnerable to transient failures. The sheer number of components required for an exascale system will all but guarantee that faults occur more frequently. The large scale parallelism will also make software bugs and human error more likely.

Several improvements to our protocol are possible. One is to make the loggers more reliable, for example by running them on redundant nodes. This would require few extra nodes, but would virtually eliminate the need for full application checkpoints and restarts. Because most failures cause only a single node to restart, it is possible to increase the checkpoint interval. This has the disadvantage that the message payload logs may grow too big, but would further lower the overhead of our protocol. Finally, since we already are logging messages, we could let nodes do individual checkpoints between the regularly scheduled coordinate checkpoints. Those individual checkpoints could be triggered when a message log on a node grows beyond a threshold.

### A. Validation

We use Daly's equation to model coordinated checkpoint/restart. It has been validated in [4]. For the double redundant case we use equation 1 from [5]. It is an approximation and has been validated in [5] against a couple of simulators and empirically. We validated simulation of the combined protocol up to 1,024 MPI ranks using our partial implementation. We also have a high level analytical model of the combined protocol:

$$E \approx \frac{W}{1 - \frac{\tau}{1.5\Theta} - \frac{R}{\Theta}}$$

where $E$ is elapsed time, $W$ is failure-free time to solution (work to be done), $\tau$ is checkpoint interval, $\Theta$ is system MTBF, and $R$ is socket restart time. Up to 100,000 sockets the simulation and model differ by less than 4%.

The rudimentary model becomes less accurate at higher socket counts due to its ignorance of full application restarts, which occur more often at high socket counts. The model assumes failures are uniformly distributed, not exponential, and does not account for writing checkpoints.

## VI. Related work

Checkpoint based techniques are the subject of numerous studies. Two different families of such techniques exist: methods that coordinate checkpoints of all processes in order to create a consistent snapshot of the system, and methods that log message content and information about the reception, without coordinating checkpoints between processes. Both families are detailed in [2]. In this Section we present recent advances in this domain and differences to our approach.

### A. Coordinated checkpoints

Our protocol relies on blocking, coordinated checkpoints. This technique consists of taking regular snapshots of the global system state [14] by coordinating process checkpoints. When a process fails, the whole application is then restarted from the last snapshot taken. Two main techniques are used to coordinate checkpoints. One, called *nonblocking*, makes use of the snapshot algorithm [14] and lets the application execute while transparently checkpointing the processes states. The other, called *blocking*, quiesces the network prior to checkpointing the system's state, avoiding communication during that time and, thus, impacting the application performance.

Blocking techniques are used in LAM/MPI [15] and Open MPI [16] because they are agnostic of the underlying network architecture. During a blocking checkpoint, the network is quiescent, and NIC state does not need to be saved, making it possible to restart on a node with a different network type. A study comparing blocking and nonblocking techniques [17] demonstrates that both of these two approaches do not scale well. Moreover, an increased frequency of process failures can lead to application never terminating [18]. Contrary to these

approaches, our algorithm uses message logging techniques which usually prevent the whole application from restarting.

### B. Pessimistic, causal, and optimistic message logging

Most of the message logging techniques do not coordinate checkpoints. Instead, these protocols rely on piecewise determinism [2]. The execution of a process is assumed to be determined by its sequence of message receptions. These protocols keep logs of message events and store message payloads in either a remote stable storage [19] or in the sender's volatile memory [20]. When a process restarts from a previous checkpoint, it will receive the exact same messages, in the same order, again. This is necessary for consistency.

In [2], [21], message logging protocols are formally categorized into three classes: pessimistic, causal, and optimistic. Pessimistic protocols ensure all information needed for re-executing a process is stored in stable storage before a process can receive another message. Thus, these protocols have a high impact on message rate [19], [22].

Causal protocols ensure that the information needed for system consistency after a process fails, is either stored in stable storage or in the volatile memory of some other processes. To achieve this, causal protocols piggyback information on every message. This diminishes the impact on latency but affects bandwidth due to the often large amount of additional information in each message [18].

Optimistic message logging protocols [23]–[25] let applications continue their execution without making sure all information will be recoverable. These protocols can therefore create orphans which can cause inconsistent system state. Optimistic protocols use various mechanisms to detect such situations and to find a consistent snapshot from all previous process checkpoints. This can potentially lead to an application having to restart from the beginning. Following this classification scheme for log-based checkpoint/restart, the logging portion of our protocol can be described as optimistic and is most similar to [3]. Contrary to existing optimistic protocols, when the rare situation of creating orphans occurs, we can use the last coordinated checkpoint to recover the application. Another difference is that we do not assume event loggers to use stable storage. Message logging protocols store message payload and event information. Garbage collection is needed to cleanup these logs. In our case, that is simple. When a coordinated checkpoint finishes, or if a full application restart is required, message payload and event information is erased.

In this paper, we do not delve deeply into the exact way of how logging is to be done. For example, [26] discusses that MPI messages are not monolithic blocks of data, and that modern NICs make the determination of when a message has been delivered more complex. [27] present techniques to reduce the cost of memory copies needed to log message payloads. There is also previous work looking at specialized hardware to serve as logging memory or stable storage. Examples are [28] which investigates PCRAM for the log memory, and [29] which looks at non-volatile memory for stable storage. These optimizations are out of scope for our study.

### C. Similar work

Combining message logging with coordinated checkpointing has been proposed in [30]. Different from our approach, that work relies on pessimistic message logging and stable storage. Using a pessimistic technique is more costly than an optimistic one, but prevents creation of orphans and thus prevents having to restart the whole application. In [30], coordinated checkpointing is used for garbage collection and to prevent storing message payloads in stable storage. A recent study [26] indicates that pessimistic message logging may have too high of a cost for applications at exascale.

Work that is closely related to ours is described in [31]. That paper explains that coordinated checkpoint/restart does not scale and that the alternative – message logging – has the problem of high storage overhead, particularly on nodes with a high core count. Because failures among cores of the same node are correlated, Bouteiller et al. propose a hybrid method: coordinated within nodes and uncoordinated with message logging between nodes.

There have been several attempts to reduce the overhead of message logging. Examples include [32] and [33]. The latter makes use of send deterministic properties, which many MPI HPC applications possess, to avoid checkpoint coordination and to lower the amount of logging data. Our approach is limited to certain types of applications as well. There have also been attempts to improve checkpoint/restart without message logging; e.g., [34]. Our approach differs because we do use message logging, but do not incur most of the overheads traditionally associated with message logging. Work like [18], [35] and [36] compare the two approaches with each other, but do not consider the combination of the two.

## VII. Conclusions

We investigated a combination of optimistic message logging that uses external loggers in combination with coordinated checkpoint/restart for its suitability to make a class of exascale applications more fault tolerant. We found that this particular combination of techniques allows an application to complete its work in a total time that rivals the time that would be required in a redundant system, but with a small fraction of the additional nodes required for redundant computing.

The protocol presented here is not a universal solution to fault tolerance at exascale. Some applications, for example those that are already memory-size constrained, may not be able to accommodate logs. In those situations, an approach that uses additional hardware, for example redundant computing, may be the preferred solution. However, in this paper we have shown that combining coordinated checkpointing with optimistic message logging is performance compatible with redundant computing, but requires far fewer additional nodes.

## References

[1] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill, "Recent advances in checkpoint/recovery systems," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, Apr. 2006.

[2] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[3] T. Ropars and C. Morin, "Active optimistic message logging for reliable execution of MPI applications," in *15th International Euro-Par Conference*, Delft, Netherlands, Aug. 2009.

[4] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Gener. Comput. Syst.*, vol. 22, no. 3, pp. 303–312, 2006.

[5] K. Ferreira, R. Riesen, P. G. Bridges, D. Arnold, Steraley, J. H. L. III, R. Oldfield, K. Pedretti, and R. Brightwell, "Evaluating the viability of process replication reliability for exascale systems," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[6] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, P. Kogge, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," http://www.science.energy.gov/ascr/Research/CS/DARPAexascale-hardware(2008).pdf, Sep. 2008.

[7] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in *International Conference on Dependable Systems and Networks (DSN)*, Jun. 2006.

[8] T. J. Hacker, F. Romero, and C. D. Carothers, "An analysis of clustered failures on large supercomputing systems," *J. Parallel Distrib. Comput.*, vol. 69, pp. 652–665, Jul. 2009.

[9] J. E. S. Hertel, R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. PetneY, S. A. Silling, P. A. Taylor, and L. Yarrington, "CTH: A software family for multi-dimensional shock physics analysis," in *Proceedings of the 19th International Symposium on Shock Waves*, Jul. 1993, pp. 377–382.

[10] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings, "Predictive performance and scalability modeling of a large-scale application," in *Proceedings of the ACM/IEEE conference on Supercomputing*, 2001, pp. 37–48.

[11] S. J. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J Comp Phys*, vol. 117, no. 1, pp. 1–19, 1995.

[12] Sandia National Laboratory, "LAMMPS molecular dynamics simulator," http://lammps.sandia.gov, Apr. 10 2010.

[13] ——, "Mantevo project home page," https://software.sandia.gov/mantevo, Apr. 10 2010.

[14] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, 1985.

[15] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The LAM/MPI checkpoint/restart framework: System-initiated checkpointing," *International Journal of High Performance Computing Applications*, vol. 19, no. 4, pp. 479–493, Winter 2005.

[16] J. Hursey, T. I. Mattox, and A. Lumsdaine, "Interconnect agnostic checkpoint/restart in Open MPI," in *HPDC '09: Proceedings of the 18th ACM international symposium on High Performance Distributed Computing*. New York, NY, USA: ACM, 2009, pp. 49–58.

[17] D. Buntinas, C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI protocols," *Future Generation Computer Systems*, vol. 24, no. 1, pp. 73–84, 2008.

[18] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello, "Improved message logging versus improved coordinated checkpointing for fault tolerant MPI," in *IEEE International Conference on Cluster Computing*, 2004, pp. 115–124.

[19] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '02. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002.

[20] D. B. Johnson and W. Zwaenepoel, "Sender-based message logging," in *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*, 1987.

[21] L. Alvisi and K. Marzullo, "Message logging: Pessimistic, optimistic, causal, and optimal," *IEEE Trans. Softw. Eng.*, vol. 24, no. 2, pp. 149–159, Feb. 1998.

[22] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette, "Mpich-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2003.

[23] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using asynchronous and checkpointing," in *Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1988, pp. 171–181.

[24] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Comput. Surv.*, vol. 37, no. 1, pp. 42–81, 2005.

[25] Q. Jiang, Y. Luo, and D. Manivannan, "An optimistic checkpointing and message logging approach for consistent global checkpoint collection in distributed systems," *J. Parallel Distrib. Comput.*, vol. 68, no. 12, pp. 1575–1589, 2008.

[26] A. Bouteiller, T. Ropars, G. Bosilca, C. Morin, and J. Dongarra, "Reasons for a pessimistic or optimistic message logging protocol in MPI uncoordinated failure recovery," in *IEEE International Conference on Cluster Computing*, Sep. 2009.

[27] B. George, B. Aurelien, H. Thomas, L. Pierre, and D. J. J., "Dodging the cost of unavoidable memory copies in message logging protocols," in *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, ser. EuroMPI'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 189–197.

[28] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2009.

[29] M. Banâtre, A. Gefflaut, P. Joubert, C. Morin, and P. Lee, "An architecture for tolerating processor failures in shared-memory multiprocessors," *Computers, IEEE Transactions on*, vol. 45, no. 10, pp. 1101–1115, Oct. 1996.

[30] E. N. Elnozahy and W. Zwaenepoel, "On the use and implementation of message logging," in *Digest of Papers: FTCS/24, The Twenty-Fourth Annual International Symposium on Fault-Tolerant Computing, Austin, Texas, USA, June 15-17, 1994*. IEEE Computer Society, 1994, pp. 298–307.

[31] A. Bouteiller, T. Herault, G. Bosilca, and J. J. Dongarra, "Correlated set coordination in fault tolerant message logging protocols," in *17th International Conference on Parallel Processing*, 2011, pp. 51–64.

[32] A. Bouteiller, G. Bosilca, and J. Dongarra, "Redesigning the message logging model for high performance," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 16, pp. 2196–2211, 2010.

[33] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic MPI applications," in *International Parallel Distributed Processing Symposium (IPDPS)*, may 2011, pp. 989–1000.

[34] K. Z. Meth and W. G. Tuel, "Parallel checkpoint/restart without message logging," in *International Workshop on Parallel Processing*, 2000, pp. 253–258.

[35] P. Lemarinier, A. Bouteiller, G. Krawezik, and F. Cappello, "Coordinated checkpoint versus message log for fault tolerant MPI," *Int. J. High Perform. Comput. Netw.*, vol. 2, no. 2-4, pp. 146–155, 2004.

[36] A. Bouteiller, P. Lemarinier, K. Krawezik, and F. Capello, "Coordinated checkpoint versus message log for fault tolerant MPI," in *IEEE International Conference on Cluster Computing*, 2003, pp. 242–250.